


GALAHAD QPB

USER DOCUMENTATION GALAHAD Optimization Library version 5.1

1 SUMMARY

This package uses an primal-dual interior-point trust-region method to solve the **quadratic programming problem**

$$\text{minimize } \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f$$

subject to the general linear constraints

$$c_i^l \leq \mathbf{a}_i^T\mathbf{x} \leq c_i^u, \quad i=1,\ldots,m,$$

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j=1,\ldots,n,$$

where the n by n symmetric matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the scalar f are given. Full advantage is taken of any zero coefficients in the matrix $\mathbf{H}$ or the vectors $\mathbf{a}_i$. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite.

If the matrix $\mathbf{H}$ is positive semi-definite, a global solution is found. However, if $\mathbf{H}$ is indefinite, the procedure may find a (weak second-order) critical point that is not the global solution to the given problem.

ATTRIBUTES — Versions: `GALAHAD_QPB_single`, `GALAHAD_QPB_double`. **Uses:** `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NORMS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_TOOLS`, `GALAHAD_ROOTS`, `GALAHAD_QPT`, `GALAHAD_QPP`, `GALAHAD_QPD`, `GALAHAD_LSQP`, `GALAHAD_SBLS`, `GALAHAD_FDC`, `GALAHAD_GLTR`, `GALAHAD_FIT`. **Date:** December 1999. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_QPB_single
```

with the obvious substitution `GALAHAD_QPB_double`, `GALAHAD_QPB_quadruple`, `GALAHAD_QPB_single_64`, `GALAHAD_QPB_double_64` and `GALAHAD_QPB_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `QPB_time_type`, `QPB_control_type`, `QPB_inform_type` and `QPB_data_type` (Section 2.4) and the subroutines `QPB_initialize`, `QPB_solve`, `QPB_terminate`, (Section 2.5) and `QPB_read_specfile` (Section 2.7) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Both the Hessian matrix $\mathbf{H}$ and the constraint Jacobian $\mathbf{A}$, the matrix whose rows are the vectors $\mathbf{a}_i^T$, $i=1,\ldots,m$, may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix **A** is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n*(i-1)+j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1,\ldots,m$, $j = 1,\ldots,n$. Since **H** is symmetric, only the lower triangular part (that is the part h_{ij} for $1 \leq j \leq i \leq n$) need be held. In this case the lower triangle will be stored by rows, that is component $i*(i-1)/2+j$ of the storage array `H%val` will hold the value h_{ij} (and, by symmetry, h_{ji}) for $1 \leq j \leq i \leq n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **A**, its row index i, column index j and value a_{ij} are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to **H** (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val` and an integer value `H%ne`), except that only the entries in the lower triangle need be stored.

2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of **A**, the i-th component of a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr` $(m+1)$ holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i-th row are stored in components $l =$ `A%ptr`(i), ...,`A%ptr` $(i+1)-1$ of the integer array `A%col`, and real array `A%val`, respectively. The same scheme is applicable to **H** (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle need be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.1.4 Diagonal storage format

If **H** is diagonal (i.e., $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonals entries h_{ii}, $1 \leq i \leq n$, need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square **A**.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.3 Parallel usage

OpenMP may be used by the `GALAHAD_QPB` package to provide parallelism for some solvers in shared memory environments. See the documentation for the GALAHAD package `SLS` for more details. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of `-openmp`). The number of threads may be controlled at runtime by setting the environment variable `OMP_NUM_THREADS`.

MPI may also be used by the package to provide parallelism for some solvers in a distributed memory environment. To use this form of parallelism, MPI must be enabled at runtime by using the correct compiler flag (usually some variant of `-lmpi`). Although the MPI process will be started automatically when required, it should be stopped by the

calling program once no further use of this form of parallelism is needed. Typically, this will be via statements of the form

```
CALL MPI_INITIALIZED( flag, ierr )
IF ( flag ) CALL MPI_FINALIZE( ierr )
```

The code may be compiled and run in serial mode.

2.4 The derived data types

Six derived data types are accessible from the package.

2.4.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrices **A** and **H**. The components of SMT_TYPE used here are:

m is a scalar component of type INTEGER(ip_), that holds the number of rows in the matrix.

n is a scalar component of type INTEGER(ip_), that holds the number of columns in the matrix.

ne is a scalar variable of type INTEGER(ip_), that holds the number of matrix entries.

type is a rank-one allocatable array of type default CHARACTER, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.4.2).

val is a rank-one allocatable array of type REAL(rp_) and dimension at least ne, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix **H** is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the row indices of the entries. (see §2.1.2).

col is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the column indices of the entries (see §2.1.2–2.1.3).

ptr is a rank-one allocatable array of type INTEGER(ip_), and dimension at least m + 1, that may hold the pointers to the first entry in each row (see §2.1.3).

2.4.2 The derived data type for holding the problem

The derived data type QPT_problem_type is used to hold the problem. The components of QPT_problem_type are:

new_problem_structure is a scalar variable of type default LOGICAL, that is .TRUE. if this is the first (or only) problem in a sequence of problems with identical "structure" to be attempted, and .FALSE. if a previous problem with the same "structure" (but different numerical data) has been solved. Here, the term "structure" refers both to the sparsity patterns of the Jacobian matrices **A** involved (but not their numerical values), to the zero/nonzero/infinity patterns (a bound is either zero, $\pm$ infinity, or a finite but arbitrary nonzero) of each of the constraint bounds, and to the variables and constraints that are fixed (both bounds are the same) or free (the lower and upper bounds are $\pm$ infinity, respectively).

n is a scalar variable of type INTEGER(ip_), that holds the number of optimization variables, n.

m is a scalar variable of type INTEGER(ip_), that holds the number of general linear constraints, m.

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H**. The following components are used:

H%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of H%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of H%type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of H%type must contain the string SPARSE_BY_ROWS, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of H%type must contain the string DIAGONAL.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into H%type. For example, if prob is of derived type QPB_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( prob%H%type, 'COORDINATE', istat )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

H%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other three schemes.

H%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.1.

H%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other three schemes.

H%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of the **lower triangular** part of **H** in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

H%ptr is a rank-one allocatable array of dimension n+1 and type INTEGER(ip_), that holds the starting position of each row of the **lower triangular** part of **H**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

G is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the gradient $\mathbf{g}$ of the linear term of the quadratic objective function. The j-th component of G, $j = 1,\ldots,n$, contains $\mathbf{g}_j$.

f is a scalar variable of type REAL(rp_), that holds the constant term, f, in the objective function.

A is scalar variable of type SMT_TYPE that holds the Jacobian matrix **A**. The following components are used:

A%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of A%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of A%type must contain the string COORDINATE, while for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of A%type must contain the string SPARSE_BY_ROWS.

Just as for H%type above, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into A%type. Once again, if prob is of derived type QPB_problem_type and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

```
CALL SMT_put( prob%A%type, 'SPARSE_BY_ROWS', istat )
```

A%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for either of the other two schemes.

A%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix **A** in any of the storage schemes discussed in Section 2.1.

A%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.

A%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **A** in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense storage scheme is used.

A%ptr is a rank-one allocatable array of dimension m+1 and type INTEGER(ip_), that holds the starting position of each row of **A**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see Section 2.1.3). It need not be allocated when the other schemes are used.

C_l is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i-th component of C_l, $i = 1,\ldots,m$, contains $\mathbf{c}^l_i$. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.4.3).

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains $\mathbf{c}^u_i$. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than infinity, where infinity is a component of the control array control (see Section 2.4.3).

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}^l_j$. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.4.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}^u_j$. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.4.3).

X is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the values **x** of the optimization variables. The j-th component of X, $j = 1,\ldots,n$, contains x_j.

Z is a rank-one allocatable array of dimension n and type default REAL(rp_), that holds the values **z** of estimates of the dual variables corresponding to the simple bound constraints (see Section 4). The j-th component of Z, $j = 1,\ldots,n$, contains z_j.

C is a rank-one allocatable array of dimension m and type default REAL(rp_), that holds the values **Ax** of the constraints. The i-th component of C, $i = 1,\ldots,m$, contains $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$.

Y is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the values **y** of estimates of the Lagrange multipliers corresponding to the general linear constraints (see Section 4). The i-th component of Y, $i = 1,\ldots,m$, contains y_i.

2.4.3 The derived data type for holding control parameters

The derived data type QPB_control_type is used to hold controlling data. Default values may be obtained by calling QPB_initialize (see Section 2.5.1), while components may also be changed by calling GALAHAD_QPB_read_spec (see Section 2.7.1). The components of QPB_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in QPB_solve and QPB_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the stream number for informational messages. Printing of informational messages in QPB_solve is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

start_print is a scalar variable of type INTEGER(ip_), that specifies the first iteration for which printing will occur in QPB_solve. If start_print is negative, printing will occur from the outset. The default is start_print = -1.

stop_print is a scalar variable of type INTEGER(ip_), that specifies the last iteration for which printing will occur in QPB_solve. If stop_print is negative, printing will occur once it has been started by start_print. The default is stop_print = -1.

maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of iterations which will be allowed in QPB_solve. The default is maxit = 1000.

infeas_max is a scalar variable of type INTEGER(ip_), that specifies the number of iterations for which the overall infeasibility of the problem is not reduced by at least a factor reduce_infeas before the problem is flagged as infeasible (see reduce_infeas). The default is infeas_max = 200.

cg_maxit is a scalar variable of type INTEGER(ip_), that holds the maximum number of conjugate-gradient inner iterations that may be performed during the computation of each search direction in QPB_solve. If cg_maxit is set to a negative number, it will be reset by QPB_solve to the dimension of the relevant linear system $+1$. The default is cg_maxit = 200.

indicator_type is a scalar variable of type INTEGER(ip_), that specifies the type of indicator used to assess when a variable or constraint is active. Possible values are:

- 1 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_p (see below).
- 2 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_pd (see below) times the magnitude of its corresponding dual variable.
- 3 a variable/constraint is active if and only if the distance to its neaerest bound is no larger than indicator_tol_tapia (see below) times the distance to the same bound at the previous iteration.

The default is indicator_type = 3.

restore_problem is a scalar variable of type INTEGER(ip_), that specifies how much of the input problem is to be retored on output. Possible values are:

- 0 nothing is restored.
- 1 the vector data $\mathbf{g}$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, and $\mathbf{x}^u$ will be restored to their input values.
- 2 the entire problem, that is the above vector data along with the Hessian matrix $\mathbf{H}$ and the Jacobian matrix $\mathbf{A}$, will be restored.

The default is restore_problem = 2.

extrapolate is a scalar variable of type INTEGER(ip_), that specifies whether extrapolation should be used to track the central path. Possible values are:

- 0 no extrapolation should be used.
- 1 extrapolation should be used after the final update of the barrier parameter.
- 2 extrapolation should be used after each update of the barrier parameter.

The default is extrapolate = 2.

path_history is a scalar variable of type INTEGER(ip_), that specifies the maximum number of previous path points used when extrapolating to track the central path. The default is path_history = 1.

path_derivatives is a scalar variable of type INTEGER(ip_), that specifies the maximum order of derivatives to use when extrapolating to track the central path. The default is path_derivatives = 5.

fit_order is a scalar variable of type INTEGER(ip_), that specifies the order of (Puiseux or Taylor) series to fit to the available data when building the extrapolant to track the central path. If fit_order $\leq$ 0, the order will be such that all available data as specified by path_history and path_derivatives will be used. The default is fit_order = - 1.

infinity is a scalar variable of type REAL(rp_), that is used to specify which constraint bounds are infinite. Any bound larger than infinity in modulus will be regarded as infinite. The default is infinity = 10^{19}.

stop_p is a scalar variable of type REAL(rp_), that holds the required accuracy for the primal infeasibility (see Section 4). The default is stop_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

stop_d is a scalar variable of type default REAL(rp_), that holds the required accuracy for the dual infeasibility (see Section 4). The default is stop_d = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

stop_c is a scalar variable of type default REAL(rp_), that holds the required accuracy for the violation of complementarity slackness (see Section 4). The default is stop_c = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

prfeas is a scalar variable of type REAL(rp_), that aims to specify the closest that any initial variable may be to infeasibility. Any variable closer to infeasibility than prfeas will be moved to prfeas from the offending bound. However, if a variable is range bounded, and its bounds are closer than prfeas apart, it will be moved to the mid-point of the two bounds. The default is prfeas = 1.0.

dufeas is a scalar variable of type REAL(rp_), that aims to specify the closest that any initial dual variable or Lagrange multiplier may be to infeasibility. Any variable closer to infeasibility than prfeas will be moved to dufeas from the offending bound. However, if a dual variable is range bounded, and its bounds are closer than dufeas apart, it will be moved to the mid-point of the two bounds. The default is dufeas = 1.0.

muzero is a scalar variable of type REAL(rp_), that holds the initial value of the barrier parameter. If muzero is not positive, it will be reset automatically to an appropriate value. The default is muzero = -1.0.

reduce_infeas is a scalar variable of type default REAL(rp_), that specifies the least factor by which the overall infeasibility of the problem must be reduced, over infeas_max consecutive iterations, for it not be declared infeasible (see infeas_max). The default is reduce_infeas = 0.99.

obj_unbounded is a scalar variable of type default REAL(rp_), that specifies smallest value of the objective function that will be tolerated before the problem is declared to be unbounded from below. The default is obj_unbounded = $-u^{-2}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

identical_bounds_tol is a scalar variable of type REAL(rp_). Every pair of constraint bounds (c_i^l, c_i^u) or (x_j^l, x_j^u) that is closer than identical_bounds_tol will be reset to the average of their values, $\frac{1}{2}(c_i^l + c_i^u)$ or $\frac{1}{2}(x_j^l + x_j^u)$ respectively. The default is identical_bounds_tol = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

indicator_tol_p is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 1. The default is indicator_tol_p = $u^{1/3}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

indicator_tol_pd is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 2. The default is indicator_tol_pd = 1.0.

indicator_tol_tapia is a scalar variable of type REAL(rp_) that provides the indicator tolerance associated with the test indicator_type = 3. The default is indicator_tol_tapia = 0.9.

inner_stop_relative and inner_stop_absolute are scalar variables of type REAL(rp_), that hold the relative and absolute convergence tolerances for the inner iteration (search direction) problem using the package GALAHAD_GLTR, and correspond to the values control%stop_relative and control%stop_absolute in that package. The defaults are inner_stop_relative = 0.01 and inner_stop_absolute = $\sqrt{u}$, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_QPB_double).

initial_radius is a scalar variable of type default REAL(rp_), that specifies the initial trust-region radius. If initial_radius is not positive, it will be reset automatically to an appropriate value. The default is initial_radius = - 1.0.

inner_fraction_opt is a scalar variable of type default REAL(rp_), that specifies the fraction of the optimal value which is acceptable for the solution of the inner iteration (search direction) problem using the package GALAHAD_GLTR, and corresponds to the value control%fraction_opt in that package. A negative value is considered to be zero, and a value of larger than one is considered to be one. Reducing fraction_opt below one will result in a reduction of the computation performed at the expense of an inferior optimal value. The default is inner_fraction_opt = 0.1.

cpu_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted CPU time. Any negative value indicates no limit will be imposed. The default is cpu_time_limit = - 1.0.

clock_time_limit is a scalar variable of type REAL(rp_), that is used to specify the maximum permitted elapsed system clock time. Any negative value indicates no limit will be imposed. The default is clock_time_limit = - 1.0.

remove_dependencies is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm is to attempt to remove any linearly dependent constraints before solving the problem, and .FALSE. otherwise. We recommend removing linearly dependencies. The default is remove_dependencies = .TRUE..

treat_zero_bounds_as_general is a scalar variable of type default LOGICAL. If it is set to .FALSE., variables which are only bounded on one side, and whose bound is zero, will be recognised as non-negativities/non-positivities rather than simply as lower- or upper-bounded variables. If it is set to .TRUE., any variable bound x_j^l or x_j^u which has the value 0.0 will be treated as if it had a general value. Setting treat_zero_bounds_as_general to .TRUE. has the advantage that if a sequence of problems are reordered, then bounds which are "accidentally" zero will be considered to have the same structure as those which are nonzero. However, GALAHAD_QPB is able to take special advantage of non-negativities/non-positivities, so if a single problem, or if a sequence of problems whose bound structure is known not to change, is/are to be solved, it will pay to set the variable to .FALSE.. The default is treat_zero_bounds_as_general = .FALSE..

center is a scalar variable of type default LOGICAL, that must be set .TRUE. if the algorithm will should use the analytic center of the feasible set as its initial feasible point, and .FALSE. otherwise. We recommend using the analytic center. The default is center = .TRUE..

primal is a scalar variable of type default LOGICAL, that must be set .TRUE. if a primal-barrier Hessian will be used and .FALSE. if the primal-dual Hessian is preferred. We recommend using the primal-dual Hessian. The default is primal = .FALSE..

puiseux is a scalar variable of type default LOGICAL, that must be set .TRUE. if a Puiseux series will be used when extrapolating along the central path and .FALSE. if a Taylor series is preferred. We recommend using the Puiseux series unless the solution is known to be non-degenerate. The default is puiseux = .TRUE..

feasol is a scalar variable of type default LOGICAL, that should be set .TRUE. if the final solution obtained will be perturbed so that variables close to their bounds are moved onto these bounds, and .FALSE. otherwise. The default is feasol = .FALSE..

prefix is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix = "".

LSQP_control is a scalar variable of type LSQP_control_type whose components are used to control the initial feasible point calculation performed by the package GALAHAD_LSQP. See the specification sheet for the package GALAHAD_LSQP for details, and appropriate default values (but note that default value for LSQP_control%feasol is changed to .FALSE..

FDC_control is a scalar variable of type FDC_control_type whose components are used to control any detection of linear dependencies performed by the package GALAHAD_FDC. See the specification sheet for the package GALAHAD_FDC for details, and appropriate default values.

SBLS_control is a scalar variable of type SBLS_control_type whose components are used to control preconditioning, needed to accelerate the step calculation, performed by the package GALAHAD_SBLS. See the specification sheet for the package GALAHAD_SBLS for details, and appropriate default values (but note that if SBLS_control%preconditioner = 5, the diagonal matrix used is corresponds to replacing the Hesian matrix by its barrier terms).

GLTR_control is a scalar variable of type GLTR_control_type whose components are used to control the step calculation performed by the package GALAHAD_GLTR. See the specification sheet for the package GALAHAD_GLTR for details, and appropriate default values.

2.4.4 The derived data type for holding timing information

The derived data type QPB_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of QPB_time_type are:

total is a scalar variable of type REAL(rp_), that gives the total CPU time spent in the package.

preprocess is a scalar variable of type REAL(rp_), that gives the CPU time spent reordering the problem to standard form prior to solution.

find_dependent is a scalar variable of type REAL(rp_), that gives the CPU time spent detecting and removing linearly-dependent equality constraints

analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing the required matrices prior to factorization.

factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type REAL(rp_), that gives the CPU time spent computing the search direction.

phase1_total is a scalar variable of type REAL(rp_), that gives the total CPU time spent in the initial-feasible-point phase of the package.

phase1_analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing the required matrices prior to factorization in the initial-feasible-point phase.

phase1_factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices in the initial-feasible-point phase.

phase1_solve is a scalar variable of type REAL(rp_), that gives the CPU time spent computing the search direction in the initial-feasible-point phase.

clock_total is a scalar variable of type REAL(rp_), that gives the total elapsed system clock time spent in the package.

clock_preprocess is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent reordering the problem to standard form prior to solution.

clock_find_dependent is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent detecting and removing linearly-dependent equality constraints

clock_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing the required matrices prior to factorization.

clock_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent computing the search direction.

clock_phase1_total is a scalar variable of type REAL(rp_), that gives the total elapsed system clock time spent in the initial-feasible-point phase of the package.

clock_phase1_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing the required matrices prior to factorization in the initial-feasible-point phase.

clock_phase1_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices in the initial-feasible-point phase.

clock_phase1_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent computing the search direction in the initial-feasible-point phase.

2.4.5 The derived data type for holding informational parameters

The derived data type QPB_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of QPB_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Section 2.6 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

iter is a scalar variable of type INTEGER(ip_), that gives the total number of iterations required.

factorization_status is a scalar variable of type INTEGER(ip_), that gives the return status from the matrix factorization.

factorization_integer is a scalar variable of type long INTEGER(ip_), that gives the amount of integer storage used for the matrix factorization.

factorization_real is a scalar variable of type INTEGER(int64), that gives the amount of real storage used for the matrix factorization.

nfacts is a scalar variable of type INTEGER(ip_), that gives the total number of factorizations performed.

nbacts is a scalar variable of type INTEGER(ip_), that gives the total number of backtracks performed during the sequence of linesearches.

nmods is a scalar variable of type INTEGER(ip_), that gives the total number of factorizations which were modified to ensure that the matrix is an appropriate preconditioner.

obj is a scalar variable of type REAL(rp_), that holds the value of the objective function at the best estimate of the solution found.

non_negligible_pivot is a scalar variable of type REAL(rp_), that holds the value of the smallest pivot larger than control%zero_pivot when searching for dependent linear constraints. If non_negligible_pivot is close to control%zero_pivot, this may indicate that there are further dependent constraints, and it may be worth increasing control%zero_pivot above non_negligible_pivot and solving again.

feasible is a scalar variable of type default LOGICAL, that has the value .TRUE. if the output value of **x** satisfies the constraints, and the value .FALSE. otherwise.

time is a scalar variable of type QPB_time_type whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.4.4).

LSQP_inform is a scalar variable of type LSQP_inform_type whose components are used to provide information about the initial feasible point calculation performed by the package GALAHAD_LSQP. See the specification sheet for the package GALAHAD_LSQP for details, and appropriate default values.

FDC_inform is a scalar variable of type FDC_inform_type whose components are used to provide information about any detection of linear dependencies performed by the package GALAHAD_FDC. See the specification sheet for the package GALAHAD_FDC for details.

SBLS_inform is a scalar variable of type SBLS_inform_type whose components are used to provide information about factorizations performed by the package GALAHAD_SBLS. See the specification sheet for the package GALAHAD_SBLS for details.

GLTR_inform is a scalar variable of type GLTR_inform_type whose components are used to provide information about the step calculation performed by the package GALAHAD_GLTR. See the specification sheet for the package GALAHAD_GLTR for details.

2.4.6 The derived data type for holding problem data

The derived data type QPB_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of QPB procedures. This data should be preserved, untouched, from the initial call to QPB_initialize to the final call to QPB_terminate.

2.5 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.7 for further features):

1. The subroutine QPB_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.

2. The subroutine QPB_solve is called to solve the problem.

3. The subroutine QPB_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by QPB_solve, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since QPB_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.5.1 The initialization subroutine

Default values are provided as follows:

```
CALL QPB_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type QPB_data_type (see Section 2.4.6). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type QPB_control_type (see Section 2.4.3). On exit, control contains default values for the components as described in Section 2.4.3. These values should only be changed after calling QPB_initialize.

inform is a scalar INTENT(OUT) argument of type QPB_inform_type (see Section 2.4.5). A successful call to QPB_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.6.

2.5.2 The quadratic programming subroutine

The quadratic programming solution algorithm is called as follows:

```
CALL QPB_solve( p, data, control, inform[, C_stat, B_stat] )
```

p is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.4.2). It is used to hold data about the problem being solved. For a new problem, the user must allocate all the array components, and set values for all components except p%C. p%new_problem_structure must be set .TRUE., but will have been reset to .FALSE. on exit from QPB_solve. Users are free to choose whichever of the matrix formats described in Section 2.1 is appropriate for **A** and **H** for their application—different formats may be used for the two matrices.

For a problem with the same structure as one that has just been solved, the user may set p%new_problem_structure to .FALSE., so long as QPB_terminate has not been called in the interim. The INTEGER(ip_) components must be unaltered since the previous call to QPB_solve, but the REAL(rp_) may be altered to reflect the new problem.

The components p%X, p%Y and p%Z must be set to initial estimates, $\mathbf{x}^0$, of the primal variables, **x**, Lagrange multipliers for the general constraints, **y**, and dual variables for the bound constraints, **z**, respectively. Inappropriate initial values will be altered, so the user should not be overly concerned if suitable values are not apparent, and may be content with merely setting p%X=0.0, p%Y=0.0 and p%Z=0.0. The component p%C need not be set on entry.

On exit, the components p%X, p%Y, p%Z and p%C will contain the best estimates of the primal variables **x**, Lagrange multipliers for the general constraints **y**, dual variables for the bound constraints **z**, and values of the constraints **Ax** respectively. What of the remaining problem data has been restored depends upon the input value of the control parameter control%restore_problem. The return format for a restored array component will be the same as its input format. **Restrictions:** p%n > 0, p%m ≥ 0, p%A%ne ≥ -2 and p%H%ne ≥ -2.

data is a scalar INTENT(INOUT) argument of type QPB_data_type (see Section 2.4.6). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to QPB_initialize.

control is a scalar INTENT(IN) argument of type QPB_control_type (see Section 2.4.3). Default values may be assigned by calling QPB_initialize prior to the first call to QPB_solve.

inform is a scalar INTENT(OUT) argument of type QPB_inform_type (see Section 2.4.5). A successful call to QPB_solve is indicated when the component status has the value 0. For other return values of status, see Section 2.6.

C_stat is an OPTIONAL rank-one INTENT(OUT) array argument of dimension p%m and type INTEGER(ip_), that if PRESENT indicates which of the general linear constraints are likely in the optimal working set (that is a set of active constraints with linearly independent gradients). Possible values for C_stat(i), i= 1,..., p%m, and their meanings are

- <0 the *i*-th general constraint is in the working set, on its lower bound,
- >0 the *i*-th general constraint is in the working set, on its upper bound, and
- 0 the *i*-th general constraint is not in the working set.

B_stat is an OPTIONAL rank-one INTENT(OUT) array argument of dimension p%n and type INTEGER(ip_), that that if PRESENT indicates which of the simple bound constraints are likely in the optimal working set. Possible values for B_stat(j), j= 1,..., p%n, and their meanings are

- <0 the *j*-th simple bound constraint is in the working set, on its lower bound,
- >0 the *j*-th simple bound constraint is in the working set, on its upper bound, and
- 0 the *j*-th simple bound constraint is not in the working set.

2.5.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL QPB_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type QPB_data_type exactly as for QPB_solve, which must not have been altered **by the user** since the last call to QPB_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type QPB_control_type exactly as for QPB_solve.

inform is a scalar INTENT(OUT) argument of type QPB_inform_type exactly as for QPB_solve. Only the component status will be set on exit, and a successful call to QPB_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.6.

2.6 Warning and error messages

A negative value of inform%status on exit from QPB_solve or QPB_terminate indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

- -1. An allocation error occurred. A message indicating the offending array is written on unit control%error, and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc respectively.
- -2. A deallocation error occurred. A message indicating the offending array is written on unit control%error and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc respectively.

-3. One of the restrictions prob%n > 0 or prob%m ≥ 0 or requirements that prob%A_type and prob%H_type contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'DIAGONAL' has been violated.

-4. The bound constraints are inconsistent.

-5. The constraints appear to have no feasible point.

-7. The objective function appears to be unbounded from below on the feasible set.

-9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-10. The factorization failed; the return status from the factorization package is given in the component inform%factor_status.

-11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform%factor_status.

-16. The problem is so ill-conditioned that further progress is impossible.

-17. The step is too small to make further impact.

-18. Too many iterations have been performed. This may happen if control%maxit is too small, but may also be symptomatic of a badly scaled problem.

-19. The elapsed CPU or system clock time limit has been reached. This may happen if either control%cpu_time_limit or control%clock_time_limit is too small, but may also be symptomatic of a badly scaled problem.

-23. An entry from the strict upper triangle of **H** has been specified.

2.7 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type QPB_control_type (see Section 2.4.3), by reading an appropriate data specification file using the subroutine QPB_read_specfile. This facility is useful as it allows a user to change QPB control parameters without editing and recompiling programs that call QPB.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by QPB_read_specfile must start with a "BEGIN QPB" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by QPB_read_specfile .. )
  BEGIN QPB
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by QPB_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN QPB" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN QPB SPECIFICATION
```

and

```
END QPB SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN QPB" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when QPB_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by QPB_read_specfile.

Control parameters corresponding to the components LSQP_control, SBLS_control and GLTR_control may be changed by including additional sections enclosed by "BEGIN LSQP" and "END LSQP", "BEGIN SBLS" and "END SBLS", and "BEGIN GLTR" and "END GLTR", respectively. See the specification sheets for the packages GALAHAD_LSQP, GALAHAD_SBLS and GALAHAD_GLTR for further details.

2.7.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL QPB_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type QPB_control_type (see Section 2.4.3). Default values should have already been set, perhaps by calling QPB_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.4.3) of control that each affects are given in Table 2.1.

device is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

2.8 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control%out. If control%print_level $= 1$, a single line of output will be produced for each iteration of the process. For the initial-feasible-point phase, this will include values of the current primal and dual infeasibility, and violation of complementary slackness, the feasibility-phase objective value, the current steplength, the value of the barrier parameter, the number of backtracks in the linesearch and the elapsed CPU time in seconds. Once a suitable feasible point has been found, the iteration is divided into major iterations, at which the barrier parameter is reduced, and minor iterations, and which the barrier function is approximately minimized for the current value of the barrier parameter. For the major iterations, the value of the barrier parameter, the required values of dual feasibility and violation of

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
maximum-number-of-iterations	%maxit	integer
start-print	%start_print	integer
stop-print	%stop_print	integer
maximum-poor-iterations-before-infeasible	%infeas_max	integer
maximum-number-of-cg-iterations	%cg_maxit	integer
restore-problem-on-output	%restore_problem	integer
indicator-type-used	%indicator_type	integer
extrapolate-solution	%extrapolate	integer
path-history-length	%path_history	integer
path-derivatives-used	%path_derivatives	integer
path-fit-order	%fit_order	integer
infinity-value	%infinity	real
primal-accuracy-required	%stop_p	real
dual-accuracy-required	%stop_d	real
complementary-slackness-accuracy-required	%stop_c	real
mininum-initial-primal-feasibility	%prfeas	real
mininum-initial-dual-feasibility	%dufeas	real
initial-barrier-parameter	%muzero	real
poor-iteration-tolerance	%reduce_infeas	real
minimum-objective-before-unbounded	%obj_unbounded	real
identical-bounds-tolerance	%identical_bounds_tol	real
primal-indicator-tolerance	%indicator_tol_p	real
primal-dual-indicator-tolerance	%indicator_tol_pd	real
tapia-indicator-tolerance	%indicator_tol_tapia	real
initial-trust-region-radius	%initial_radius	real
inner-iteration-fraction-optimality-required	%inner_fraction_opt	real
inner-iteration-relative-accuracy-required	%inner_stop_relative	real
inner-iteration-absolute-accuracy-required	%inner_stop_absolute	real
maximum-cpu-time-limit	%cpu_time_limit	real
maximum-clock-time-limit	%clock_time_limit	real
remove-linear-dependencies	%remove_dependencies	logical
treat-zero-bounds-as-general	%treat_zero_bounds_as_general	logical
start-at-analytic-center	%center	logical
primal-barrier-used	%primal	logical
puiseux-extrapolation	%puiseux	logical
move-final-solution-onto-bound	%feasol	logical

Table 2.1: Specfile commands and associated components of control.

complementary slackness, and the current constraint infeasibility are reported. Each minor iteration of the optimality phase results in a line giving the current dual feasibility and violation of complementary slackness, the objective function value, the ratio of predicted to achieved reduction of the objective function, the trust-region radius, the number of backtracks in the linesearch, the number of conjugate-gradient iterations taken, and the elapsed CPU time in seconds.

If `control%print_level` ≥ 2 this output will be increased to provide significant detail of each iteration. This extra output includes residuals of the linear systems solved, and, for larger values of `control%print_level`, values of the primal and dual variables and Lagrange multipliers.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `QPB_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_NORMS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_TOOLS`, `GALAHAD_ROOTS`, `GALAHAD_QPT`, `GALAHAD_QPP`, `GALAHAD_QPD`, `GALAHAD_LSQP`, `GALAHAD_SBLS`, `GALAHAD_FDC`, `GALAHAD_GLTR` and `GALAHAD_FIT`.

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: `prob%n` > 0, `prob%m` ≥ 0, `prob%A_type` and `prob%H_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

The required solution $\mathbf{x}$ necessarily satisfies the primal optimality conditions

$$\mathbf{A}\mathbf{x} = \mathbf{c} \tag{4.1}$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \ \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \tag{4.2}$$

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{A}^T\mathbf{y} + \mathbf{z}, \ \mathbf{y} = \mathbf{y}^l + \mathbf{y}^u \ \text{and} \ \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \tag{4.3}$$

and

$$\mathbf{y}^l \geq 0, \ \mathbf{y}^u \leq 0, \ \mathbf{z}^l \geq 0 \ \text{and} \ \mathbf{z}^u \leq 0, \tag{4.4}$$

and the complementary slackness conditions

$$(\mathbf{A}\mathbf{x} - \mathbf{c}^l)^T\mathbf{y}^l = 0, \ (\mathbf{A}\mathbf{x} - \mathbf{c}^u)^T\mathbf{y}^u = 0, \ (\mathbf{x} - \mathbf{x}^l)^T\mathbf{z}^l = 0 \ \text{and} \ (\mathbf{x} - \mathbf{x}^u)^T\mathbf{z}^u = 0, \tag{4.5}$$

where the vectors $\mathbf{y}$ and $\mathbf{z}$ are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise.

Primal-dual interior point methods iterate towards a point that satisfies these conditions by ultimately aiming to satisfy (4.1), (4.3) and (4.5), while ensuring that (4.2) and (4.4) are satisfied as strict inequalities at each stage. Appropriate norms of the amounts by which (4.1), (4.3) and (4.5) fail to be satisfied are known as the primal and dual

infeasibility, and the violation of complementary slackness, respectively. The fact that (4.2) and (4.4) are satisfied as strict inequalities gives such methods their other title, namely interior-point methods.

The problem is solved in two phases. The goal of the first "initial feasible point" phase is to find a strictly interior point which is primal feasible, that is that (4.1) is satisfied. The GALAHAD package GALAHAD_LSQP is used for this purpose, and offers the options of either accepting the first strictly feasible point found, or preferably of aiming for the so-called "analytic center" of the feasible region. Having found such a suitable initial feasible point, the second "optimality" phase ensures that (4.1) remains satisfied while iterating to satisfy dual feasibility (4.3) and complementary slackness (4.5). The optimality phase proceeds by approximately minimizing a sequence of barrier functions

$$\tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{g}^T\mathbf{x}+f-\mu\left[\sum_{i=1}^{m}\log(c_i-c_i^l)+\sum_{i=1}^{m}\log(c_i^u-c_i)+\sum_{j=1}^{n}\log(x_j-x_j^l)+\sum_{j=1}^{n}\log(x_j^u-x_j)\right],$$

for an approriate sequence of positive barrier parameters μ converging to zero while ensuring that (4.1) remain satisfied and that $\mathbf{x}$ and $\mathbf{c}$ are strictly interior points for (4.2). Note that terms in the above sumations corresponding to infinite bounds are ignored, and that equality constraints are treated specially.

Each of the barrier subproblems is solved using a trust-region method. Such a method generates a trial correction step $\Delta(\mathbf{x},\mathbf{c})$ to the current iterate $(\mathbf{x},\mathbf{c})$ by replacing the nonlinear barrier function locally by a suitable quadratic model, and approximately minimizing this model in the intersection of (4.1) and a trust region $\|\Delta(\mathbf{x},\mathbf{c})\|\leq\Delta$ for some appropriate strictly positive trust-region radius Δ and norm $\|\cdot\|$. The step is accepted/rejected and the radius adjusted on the basis of how accurately the model reproduces the value of barrier function at the trial step. If the step proves to be unacceptable, a linesearch is performed along the step to obtain an acceptable new iterate. In practice, the natural primal "Newton" model of the barrier function is frequently less successful than an alternative primal-dual model, and consequently the primal-dual model is usually to be preferred.

Once a barrier subproblem has been solved, extrapolation based on values and derivatives encountered on the central path is optionally used to determine a good starting point for the next subproblem. Traditional Taylor-series extrapolation has been superceded by more accurate Puiseux-series methods as these are particularly suited to deal with degeneracy.

The trust-region subproblem is approximately solved using the combined conjugate-gradient/Lanczos method implemented in the GALAHAD package GALAHAD_GLTR. Such a method requires a suitable preconditioner, and in our case, the only flexibility we have is in approximating the model of the Hessian. Although using a fixed form of preconditioning is sometimes effective, we have provided the option of an automatic choice, that aims to balance the cost of applying the preconditioner against the needs for an accurate solution of the trust-region subproblem. The preconditioner is applied using the GALAHAD matrix factorization package GALAHAD_SBLS, but options at this stage are to factorize the preconditioner as a whole (the so-called "augmented system" approach), or to perform a block elimination first (the "Schur-complement" approach). The latter is usually to be prefered when a (non-singular) diagonal preconditioner is used, but may be inefficient if any of the columns of $\mathbf{A}$ is too dense.

In order to make the solution as efficient as possible, the variables and constraints are reordered internally by the GALAHAD package GALAHAD_QPP prior to solution. In particular, fixed variables, and free (unbounded on both sides) constraints are temporarily removed.

References:

The method is described in detail in

A. R. Conn, N. I. M. Gould, D. Orban and Ph. L. Toint (1999). "A primal-dual trust-region algorithm for minimizing a non-convex function subject to general inequality and linear equality constraints". Mathematical Programming **87** 215-249.

5 EXAMPLE OF USE

Suppose we wish to minimize $\frac{1}{2}x_1^2 + x_2^2 + \frac{3}{2}x_3^2 + 4x_1x_3 + 2x_2 + 1$ subject to the the general linear constraints $1 \leq 2x_1 + x_2 \leq 2$ and $x_2 + x_3 = 2$, and simple bounds $-1 \leq x_1 \leq 1$ and $x_3 \leq 2$. Then, on writing the data for this problem as

$$\mathbf{H} = \begin{pmatrix} 1 & & 4 \\ & 2 & \\ 4 & & 3 \end{pmatrix}, \ \mathbf{g} = \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}, \ \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ -\infty \end{pmatrix} \text{ and } \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 2 \end{pmatrix},$$

and

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & \\ & 1 & 1 \end{pmatrix}, \ \mathbf{c}^l = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \text{ and } \mathbf{c}^u = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

in sparse co-ordinate format, we may use the following code:

```
! THIS VERSION: GALAHAD 2.2 - 23/04/2008 AT 16:30 GMT.
   PROGRAM GALAHAD_QPB_EXAMPLE
   USE GALAHAD_QPB_double         ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20
   TYPE ( QPT_problem_type ) :: p
   TYPE ( QPB_data_type ) :: data
   TYPE ( QPB_control_type ) :: control
   TYPE ( QPB_inform_type ) :: inform
   INTEGER :: s
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ) )
   ALLOCATE( p%X( n ), p%Y( m ), p%Z( n ) )
   p%new_problem_structure = .TRUE.           ! new structure
   p%n = n ; p%m = m ; p%f = 1.0_wp           ! dimensions & objective constant
   p%G = (/ 0.0_wp, 2.0_wp, 0.0_wp /)         ! objective gradient
   p%C_l = (/ 1.0_wp, 2.0_wp /)               ! constraint lower bound
   p%C_u = (/ 2.0_wp, 2.0_wp /)               ! constraint upper bound
   p%X_l = (/ - 1.0_wp, - infinity, - infinity /) ! variable lower bound
   p%X_u = (/ 1.0_wp, infinity, 2.0_wp /)     ! variable upper bound
   p%X = 0.0_wp ; p%Y = 0.0_wp ; p%Z = 0.0_wp ! start from zero
!  sparse co-ordinate storage format
   CALL SMT_put( p%H%type, 'COORDINATE', s )  ! Specify co-ordinate
   CALL SMT_put( p%A%type, 'COORDINATE', s )  ! storage for H and A
   ALLOCATE( p%H%val( h_ne ), p%H%row( h_ne ), p%H%col( h_ne ) )
   ALLOCATE( p%A%val( a_ne ), p%A%row( a_ne ), p%A%col( a_ne ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /) ! Hessian H
   p%H%row = (/ 1, 2, 3, 3 /)                   ! NB lower triangle
   p%H%col = (/ 1, 2, 3, 1 /) ; p%H%ne = h_ne
   p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
   p%A%row = (/ 1, 1, 2, 2 /)
   p%A%col = (/ 1, 2, 2, 3 /) ; p%A%ne = a_ne
! problem data complete
!do s = 1, 5
   CALL QPB_initialize( data, control, inform ) ! Initialize control parameters
   control%infinity = infinity                  ! Set infinity
!  control%print_level = 5
!  control%LSQP_control%print_level = 1
```

```
   p%X = 0.0_wp ; p%Y = 0.0_wp ; p%Z = 0.0_wp ! start from zero
   CALL QPB_solve( p, data, control, inform )   ! Solve problem
   IF ( inform%status == 0 ) THEN                 !  Successful return
     WRITE( 6, "( ' QPB: ', I0, ' iterations. Optimal objective value =',      &
    &       ES12.4, /, ' Optimal solution = ', ( 5ES12.4 ) )" )                &
     inform%iter, inform%obj, p%X
!write(6,*) inform%obj, p%X
   ELSE                                          !  Error returns
     WRITE( 6, "( ' QPB_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL QPB_terminate( data, control, inform )  !  delete internal workspace
!end do
   END PROGRAM GALAHAD_QPB_EXAMPLE
```

This produces the following output:

```
 QPB: 7 iterations. Optimal objective value =  5.4459E+00
 Optimal solution =  -5.4054E-02  1.1081E+00  8.9189E-01
```

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```
!  sparse co-ordinate storage format
...
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL SMT_put( p%H%type, 'SPARSE_BY_ROWS', s )  ! Specify sparse-by-row
   CALL SMT_put( p%A%type, 'SPARSE_BY_ROWS', s )  ! storage for H and A
   ALLOCATE( p%H%val( h_ne ), p%H%col( h_ne ), p%H%ptr( n + 1 ) )
   ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 3, 1 /)                     ! NB lower triangular
   p%H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row pointers
   p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
   p%A%col = (/ 1, 2, 2, 3 /)
   p%A%ptr = (/ 1, 3, 5 /)                        ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( p%H%type, 'DENSE', s )  ! Specify dense
   CALL SMT_put( p%A%type, 'DENSE', s )  ! storage for H and A
   ALLOCATE( p%H%val( n * ( n + 1 ) / 2 ) )
   ALLOCATE( p%A%val( n * m ) )
   p%H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 4.0_wp, 0.0_wp, 3.0_wp /) ! Hessian
   p%A%val = (/ 2.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian
! problem data complete
```

respectively.

If instead $\mathbf{H}$ had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for $\mathbf{H}$, and in this case we would instead

```
CALL SMT_put( prob%H%type, 'DIAGONAL', s )  ! Specify dense storage for H
ALLOCATE( p%H%val( n ) )
p%H%val = (/ 1.0_wp, 0.0_wp, 3.0_wp /) ! Hessian values
```

Notice here that zero diagonal entries are stored.